Software Development for Algorithmic Problems
Project 3 - Experimental Evaluation

Autoencoder Evaluation

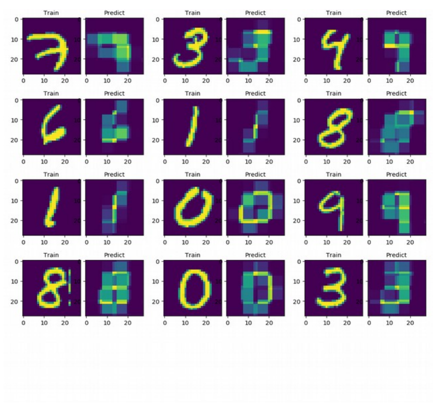
Introduction

Firstly, before proceeding into our assumptions and experimental results we should firstly define autoencoder's hyperparameters. Autoencoder's hyperparameters are the following:

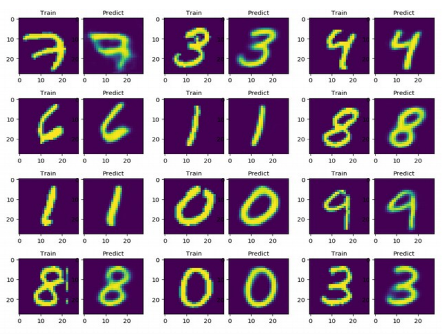
- The number of encoder's convolutional layers and the number of filters in each one of them. Also, the user can define if these layers will have afterwards a max pooling layer and a dropout layer.
- The kernel size of all the convolutional layers.
- The number of training epochs, which is how many times the neural network will read all the training set through its training.
- The batch size, which is the number of samples that the network will read before updating its weights once.
- The size of the latent vector.

Afterwards, we do the following assumptions about some of our hyperparameters. Firstly, like MNIST we have 28x28 images and we mostly take 3x3 window as kernel size as it is recommended on all tutorials, but we will also test the 5x5 kernel. Secondly, because we want to see the training progress of our autoencoder the number of epochs will be constant because if we train a model with a large number of epochs(will be 50 epochs) we can see the proper number of epochs because we will observe when the overfitting is happening. Furthermore, the convolutional layers will adopt a pyramid scheme where the first layer will have a power of two filters(base) and the next layers will have consecutive powers of two layers. For example if we have an encoder of 3 layers and a base of 16 the 3 convolutional layers will have 16, 32 and 64 filters, respectively. Finally, the metric that is used to evaluate our model is the R-squared method, although in our training process, Mean Squared Error is used with RMSProp optimizer.

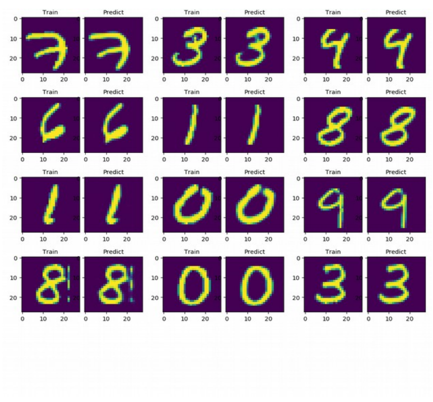
In order to provide some information and intuition to the reader in the figures below, we illustrate what each R-Squared score means in practical terms.



Autoencoder Result for R^2 equals 0.5

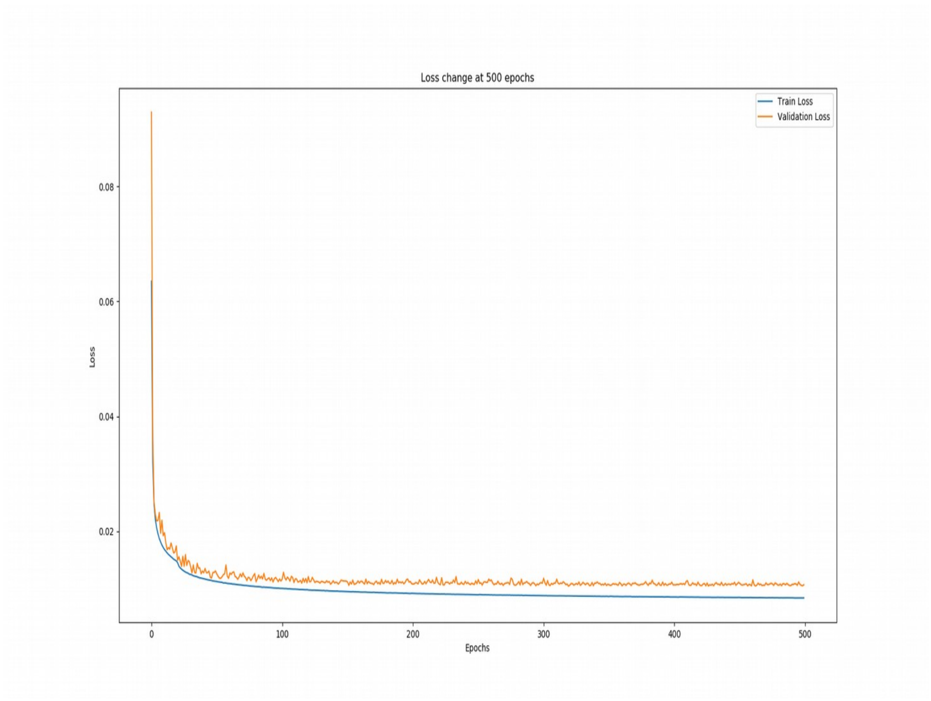


Autoencoder Result for R^2 equals 0.85



Autoencoder Result for R^2 above 0.95

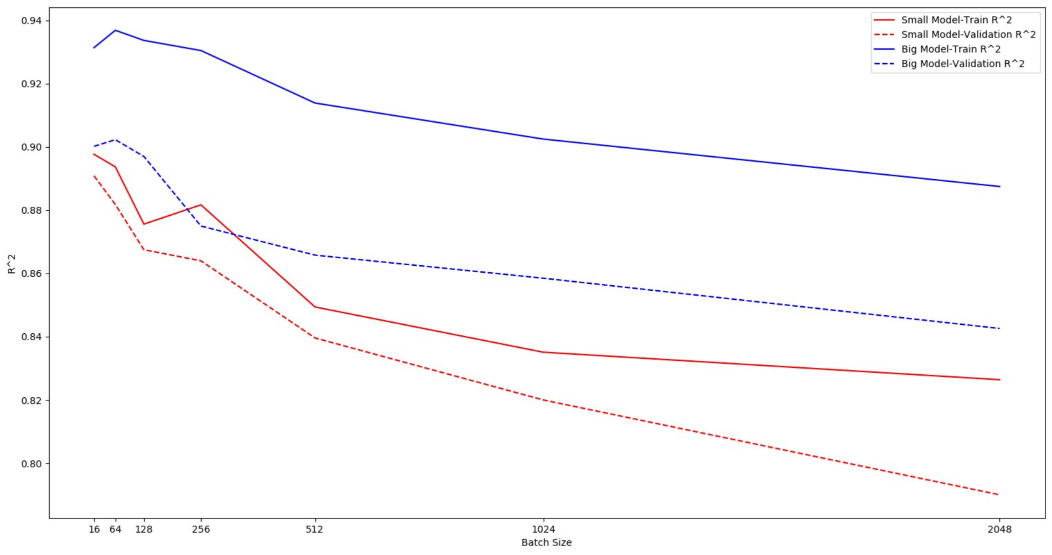
Thus, we can clearly see that when R-Squared is equal to 0.5, then the autoencoder can represent in some way the original input but the picture is unclear. When R-Squared equals 0.85 the original input is better represented, but it has some points that are not represented too precisely or in some cases might denoise the pictures but we cannot assume that because we can see at figures, pictures that are also corrupted. When R-Squared is around 0.95 where the resulting pictures are close to the original ones. Consequently, we observe that R-Squared scores greater equal 0.95 are acceptable because models with R-Squared score at 0.85 might denoise some pictures but others might get corrupted, hence they are unreliable. Note that, the better the autoencoder is, we get better latent vectors that are more capable at saving the important information of the picture given as input because the decoder is more capable at reproduccing the input image. Also, from the various experiments that are conducted we saw that this autoencoder performs worse than the autoencoder of previous assignment because of the bottleneck layer which loses much input information because it has few neurons and overfits more even that the difference between training and testing R-Squared score is a bit constant, thus we can assume that a model has a constant overfit rate.This is proven in the below figure where we train a model in terms of batch size for 50 and 500 epochs. The current model was a 3 layer with base 16, two 2x2 pools in second and third layer and batch size 256. Note that this model wasn't actually the worst but it was a model that gave decent results(around 0.88 R-Squared score) in order to perform better at less extra epochs than an even worse model.



Loss Change of Worst Model after 500 epochs

Batch Size

In this and the future experiments we test some hyperparameter using two models. The former has 3 layers(is called Small Model) and the latter 6 layers(is called Big Model). Both do not have Dropout layers because it is seem that with dropout the model performs worse. In order to test our model in respect to batch size the models have two consecutive 2x2 pooling layers with the first one being in the encoder's middle layer because this seem to have better performance(we will tell more about it afterwards). Also, the size of the bottleneck layer is 10((which is the default value). The change of R-Squared score through batch size is shown on the following graph.



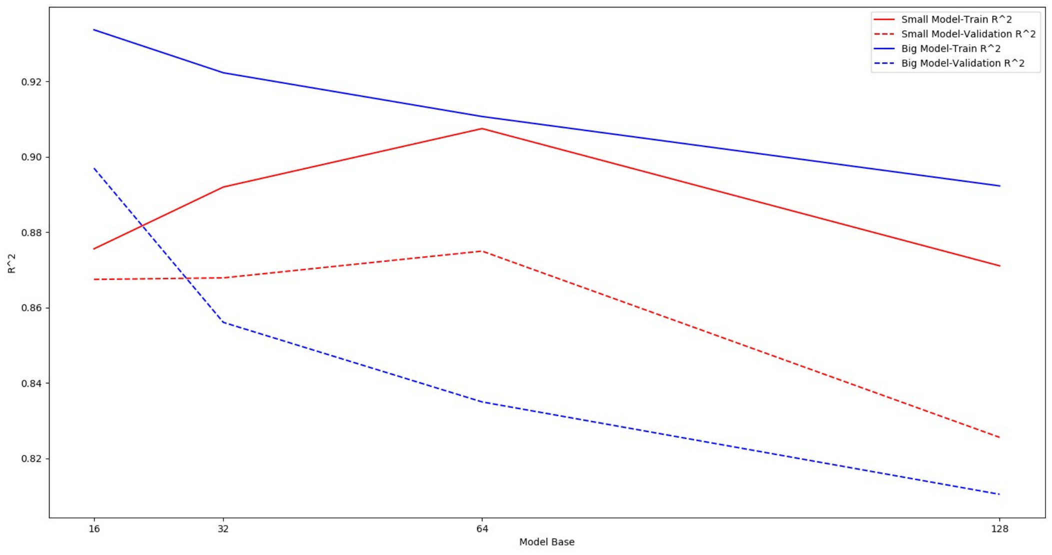
Change of R^2 in terms of batch size in Small and Big model

(Watch closely the scales on the axes)

From the above graph we can see that big model performs better than small model but it overfits more, however, still the validation R-Squared score of big model is better that the training score of small model. Thus, still big model performs better than the small one. In terms of batch size we observe that as the batch size increases our models perform worse (with one exception) and also the difference between training and validation score is bigger.

Model Base

In this experimental section we evaluate the Model's Base on Big and Small Model, with the same pooling layers, batch size of 128, and latent vectors with size 10 but with different base(until 1024, because with 2048 filters on a convolutional layer our model took too much time to train). In order for our model to not become too large we trained our models for base equal to 16, 32, 64 and 128. The results are the following.



Change of R^2 in terms of model base

From the above graph it is clear that big model performs even worse, and worse than small model with greater base and this might happen because the output of encoder's final layer might have greater dimensionality that the respective layer of small model thus the model loses more information which makes the model incapable of generalizing well. Now the small model behaves differently. It is seem that performs better and better until some value(specifically 64) and afterwards converges slower and fails more to generalize.

Dropout

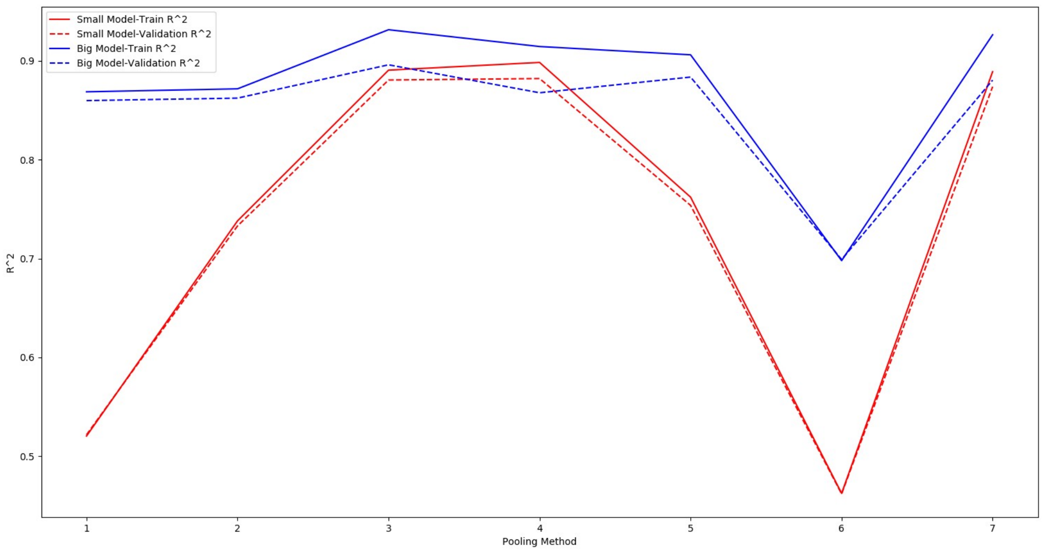
Dropout generally should be applied when the previous layer have too many neurons, because otherwise the results will be corrupted much. However, because our autoencoder performs suboptimal, we tested adding dropout but we did not see much progress, and that's why we will not test the dropout rate.

Max Pooling

Because, $28=2^2 4^1 7^1$ and the pictures have width and height equal to 28, we can have the following max pooling layers(the numbers on the enumeration are the same number on the results graph)

1. one 7x7 and two 2x2 max pooling layers
2. one 7x7 and one 2x2 max pooling layer
3. two 2x2 max pooling layers
4. one 2x2 max pooling layer
5. one 7x7 max pooling layer
6. one 7x7 and one 4x4 max pooling layer
7. one 4x4 max pooling layer

The above seven pools will be tested again on small and big model with base 16, latent vector size 10 and batch size 128. Moreover, note that that these pools are applied consecutively from the biggest pool to the smallest, beginning at the encoder's middle layer, if possible, because with these configurations we yielded better results. The results are the following:

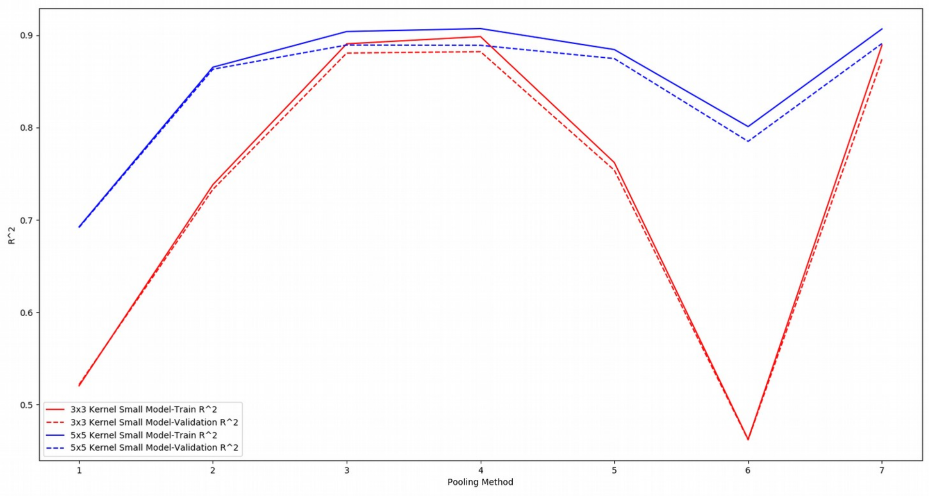


Change of R2 with respect to pooling method

From the results we observe that neither model overfits much(except the big model in some cases). However it is seem that in methods (1) and (6) where in fact the encoder returns a column vector, the big model performs better, probably because it does dimensionality reduction to more coordinates(in big into 1024 elements while in small model 64 elements). Still, it is seem that Big Model performs better than the Small one. In terms of overfit neither model seem to overfit much with only some exceptions. Finally, it is clear that the third and fourth pooling methods are the best for small model, while for big model the methods (2) and (5) are the best because in these methods the model does not overfit too much.

Kernel Size

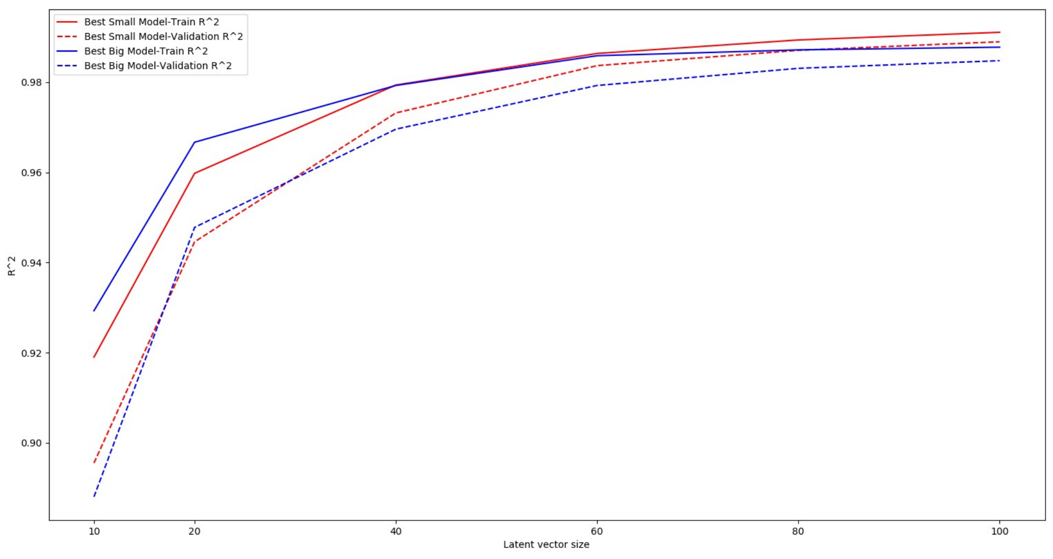
Initially, we were reluctant if it was worth to check another kernel sizes like 5x5 because we thought that either way it would produce worse results than having a kernel size of 3x3. However, we tested the 5x5 kernel size in the Small Model that we tested on pooling methods because it produced on average worse results than the Big Model and the results are the following:



We can clearly see that the performance gain in models that are using big pooling windows is huge. Also, it is seem that generally in small models having 5x5 kernel size yields better performance every time. Thus, in small model it is more wise to use kernel size equal to 5x5.

Latent vector size

This hyperparameter represents the number of neurons into the bottleneck layer. This parameter is useful because it is the target dimensionality of the input vectors and we want into these coordinates to keep the most of the input information from the input image. We will test different values of that hyperparameter to the best small and big model, providing that they have R^2 score around 0.85-0.90 at latent vector size of 10, because these models should have space for improvement. The best small model is a model with base 32, kernel size of 5x5, pooling method (4) without dropout. The best big model is a model with base 16, kernel size of 3x3, pooling method (3) without dropout. Both models will have batch size 128 and will run for 50 epochs. The results are the following:



As we can see from the above graph, both big and small model not only perform better, but also the overfitting is getting reduced. Also, it is clear that in most cases, small model performs better than small model.

Conclusion

In these part we showed how the different values of the given hyperparameters affect each model's performance. Taking all the results into consideration a good model will be a small model with latent vector size 20 or 40, batch size 128 or 256, kernel size 5x5, base 64 and having two 2x2 pools.

Nearest Neighbor Search Evaluation

Introduction

In this part we test three different methods of solving the Nearest Neighbor query. The first one is by finding the approximate Nearest Neighbor using LSH(its parameters are the same through the experiments). The second one is to do brute force search in the original space. The third one is to do brute force search to the latent vectors that were produced by the autoencoder given the input of the second method. Our experiments will be splitted into two parts. In the first part we will take latent vector of size 10 that were produced from three autoencoders, one with perfect performance(score > 0.90), one with mediocre performance(score 0.80-0.85) and one with bad performance(score 0.50) . In the second part we will take the perfect model of first part ad we will test it with latent vectors of different size.

Experiment 1

As we mentioned before in this experiment we will use the following three models:

- Bad Model: This model is a small model with base 16, batch size 128,with one 7x7 and two 2x2 pools and 3x3 kernel.
- Mediocre Model: This model is a big model with base 16, batch size 128, with one 7x7 and two 2x2 pools and 3x3 kernel.
- Perfect Model: This model is a small model with base 32, batch size 128, one 2x2 pool and 5x5 kernel.

The results of our experiments are presented in the following table:

Model/Algorithm	**Correct Predictions**	**Average Approximation Factor**	**Average Query Time(seconds)**
Brute Force	10000	1.0	0.40
LSH	6500-7500	1.03	0.040
Bad Model	1058	1.40	0.030
Mediocre Model	1410	1.29	0.030
Perfect Model	2118	1.20	0.030

As we can see from the table brute force search into the new space is faster even than LSH. However, we can see that even LSH is a little slower it outperforms by a wide margin the brute force search into the new space both into number of Correct Predictions and at Average Approximation Factor. Last but not least, even a worse model might help at denoising the pictures it is clear that the better the model the latent vectors produced are better, thus the have more correct predictions and smaller approximation factor. Thus it is seem that greater models are better at representing into a reduced space the original pictures.

Experiment 2

In this experiment we will use the perfect model of Experiment 1 we will produce latent vectors of different size and then we will run our Nearest Neighbor program. The results are the following:

Model/Latent Vector Size	**Correct Predictions**	**Average Approximation Factor**	**Average Query Time(seconds)**
Brute Force	10000	1.0	0.40
LSH	6500-7500	1.03	0.040
10	2118	1.20	0.030
20	3224	1.13	0.036
40	3554	1.11	0.045
60	3878	1.09	0.055
80	3311	1.12	0.060
100	4474	1.07	0.069

From the results above with combination of the autoencoder's results with respect to latent vector size, we can see that as the latent vector size increases both the performance of autoencoder and of the current program are getting better. However, it is clear that LSH outperforms all models and even some models with latent vector size greater than 20 the brute force search into the new space is slower than LSH, thus it is not a good choice to use latent vectors of size greater than 20 on the current problem.

Conclusion

From the above experiments it is seem clearly that search into new space is no better than LSH in any way because there is not seem a huge trade off, however someone can use it. Someone can use the perfect model with latent vectors of size 20 if he does not want to have many correct predictions and have a decent approximation factor which is a little faster than LSH, however we still do not recommend it.

EMD evaluation

General properties: Through the tests,we observe than manhattan's distance results are better than emd's.That's because in manhattan we check every pixel,so most of the times we have better results.
It also takes less time because of the complexity of the lp problem that emd requires to be solved.
Complexity for the lp is n^2 for the variables and n^2+2n for the constraints.
For 2x2 clusters we have 16 variables and 24 constraints.
For 4x4 clusters we have 256 variables and 288 constraints.
For 7x7 clusters we have 2401 variables and 2401 constraints.
For 14x14 clusters we have 38.416 variables and 38.808 constraints, which is intracable.

Emd's parameters:We observe that 2x2 clustering does not bring very good results(emd avg: 0.35),

time for 1 query and 60.000 pictures is around 9 seconds.
We observe that 4x4 clustering results are very good (emd avg: 0.87) but the time to solve the lp is increased to 72 secords for 1 query and 60.000 pictures.
We observe that 7x7 clustering clustering has the best results (emd avg: 0.91) but even more time is required for slightly better results.
For 7x7 we run the B questrion for 5000 images as well as the Emd algorithm and compared the accuracy we define below.

So, a good choice of clusters would be 4x4.

Emd compared to the brute force of question B:
Percentages of the emd's accurate results,are defined as accurate results if they have neighbor's id of B question as one of its 10 nearests neighbors.
7X7 clustering had 0.7-0.8 accuracy and and 4x4 clustering had around 0.4-0.5.
2x2 clustering has very low accuracy since the results of the emd are not very accurate.
File info has the parameters we used for the tests and output.txt has the results of each test.

Clustering Evaluation

Introduction

In this part we test three different methods of solving the Clustering problem among the MNIST images. The first one is by applying the Lloyd's algorithm to the original input. The second one is to apply Lloyd's algorithm into the new space. The third one is to take the classification results from the original images from a classifier that uses the encoding layers of last assignment's autoencoder(it is the same without the bottleneck layer)(Classes as Clusters). Our experiments will be splitted into two parts. In the first part we will take three autoencoders one with perfect performance(score > 0.90), one with mediocre performance(score 0.80-0.85) and one with bad performance(score 0.50) that all three of them will have latent vectors of size 10. These autoencoders are the same as of Nearest neighbor's search, Experiment 1. In the second part we will take the perfect model of first part ad we will test it with latent vectors of different size.

Experiment 1

In this experiment we will apply Lloyd's algorithm into latent vectors of size 10 that were produced from three models of different quality and are the same as the Nearest neighbor's experiment 1. The results are the following:

Model/Space	**Average Silhouette**	**Objective Function**	**Clustering Time(seconds)**
Original Space	0.1	1140000000	1500-2000
Classes as Clusters	0.079	1222389695	--
Worse Model	0.062	1203406404	20.63
Mediocre Model	0.065	1196872608	30.83
Best Model	0.086	1216270706	15.87

It is clear that clustering in original space outperforms all the other models while among models we cannot deduce facts because the results are irregular and there are not too much differences in performance among them.

Experiment 2

In this experiment we will apply Lloyd’s algorithm to latent vectors of different size that were produced from the best model of our previous experiment. The results are the following:

Model/Space	**Average Silhouette**	**Objective Function**	**Clustering Time(seconds)**
Original Space	0.1	1140000000	1500-2000
Classes as Clusters	0.079	1222389695	--
10	0.086	1216270706	15.87
20	0.053	1211699583	39.29
40	0.046	1246403960	136.21
60	0.043	1217573375	191.80
80	0.067	1186241650	237.74
100	0.067	1186227640	253.94

It is clear that clustering in original space outperforms all the other models while among models we cannot deduce facts because the results are irregular and there are not too much differences in performance among them. The only thing that we can deduce is that clustering on latent vectors of size 80 and 100 yield better objective function than the rest of the models.

Conclusion

From the results above we cannot conclude about what model is better, but we can assume that if someone wants to do the best possible clustering with Lloyd’s algorithm he should do it into original space, but is someone wants a faster solution he can do clustering using latent vectors of size 80 generated from the Best Model.